Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis

Christian Lindig, Gregor Snelting
Technische Universität Braunschweig
Abteilung Softwaretechnologie
Bültenweg 88
D-38104 Braunschweig, Germany
+49 531 391 7577
{lindig,snelting}@ips.cs.tu-bs.de

ABSTRACT
We apply mathematical concept analysis in order to modularize legacy code. By analysing the relation between procedures and global variables, a so-called concept lattice is constructed. The paper explains how module structures show up in the lattice, and how the lattice can be used to assess cohesion and coupling between module candidates. Certain algebraic decompositions of the lattice can lead to automatic generation of modularization proposals. The method is applied to several examples written in Modula-2, Fortran, and Cobol; among them a >100kloc aerodynamics program.

Keywords
Reengineering, Modularization, Concept Analysis

INTRODUCTION

Analysing old software has become an important topic in software technology, as there are millions of lines of legacy code which lack proper documentation; due to ongoing modifications, software entropy has increased steadily. If nothing is done, such software will die of old age — and the knowledge embodied in the software is inevitably lost. As a first step in "software geriatrics", one must reconstruct abstract concepts from the source code (called "reverse engineering"). In a second step, one might try to transform the source code such that the structure of the system is improved and obeys modern software engineering principles.

One particular problem is modularization of old code. Old systems have not been developed by today's modularization criteria. Therefore, static information like control and data flow, access to nonlocal objects, or interface information must be extracted in order to guide restructuring. Modularization can then be achieved by manual changes or automated program transformation or both (see e.g. [5]). In particular, the relation between procedures and (global) variables has long been recognized important for restructuring [13]. Indeed, an abstract data object is characterized by a set of procedures operating on a common set of (hidden) variables. Legacy systems written in FORTRAN or COBOL however make abundant use of global variables, as there is no syntactic support for modules. In old FORTRAN code for example, there is often no correspondence between COMMON blocks and modules. Thus one important step in restructuring such old systems is to discover candidates for modules or abstract data objects. Among other information sources, the relation between variables and procedures must be examined, and if possible, module candidates must be identified.

In earlier work, we have shown that *mathematical concept analysis* [15, 18] is a useful tool for analysing old software. As a particular problem, we have chosen the analysis of configurations in multi-platform source files. We have shown how dependencies and interferences between configurations can be detected using a concept lattice [8]. More recent work described how source files can be simplified according to lattice-generated information [14].

In this paper, we investigate the relation between procedures and global variables in legacy code. Based on this relation, we want to find module candidates and assess the module structure. We first formalize module structures, and give formal definitions for coupling, cohesion, and interference. We then apply mathematical concept analysis to the problem of modularizing legacy code. By analysing the relation between procedures and global variables, module candidates are identified and arranged in a so-called concept lattice. Hierarchical clustering of local modules or procedures shows up as sub-/superconcept relation in the lattice. Specific infima (so-called interferences) correspond to violations of modular structure, and proposals for interference resolution can be automatically generated. Furthermore, module candidates can sometimes be generated from certain algebraic decompositions of the lattice.

ICSE 97 Boston MA USA

FORMALIZATION OF MODULE STRUCTURES

It is our goal to find modules in legacy code by analysing the relation between procedures and global variables. We begin with some basic definitions.

Definition. Let a program consist of a set of procedures $\mathcal{P}$ and a set of variables $\mathcal{V}$. The *variable usage table* is a relation $C \subseteq \mathcal{P} \times \mathcal{V}$. If $(p, v) \in C$, procedure p uses variable v.

The variable usage table is constructed by a frontend; it is based on actual usage of global variables in procedures. Procedures and variables are assumed to be globally unique; if necessary, the frontend must provide unique names.

Definition. An abstract data object (*ADO*, or module) consists of a set of procedures $P \subseteq \mathcal{P}$ and a set of variables $V \subseteq \mathcal{V}$ such that $\forall v \in \mathcal{V}, p \in P : (p, v) \in C \Rightarrow v \in V$ and $\forall p \in \mathcal{P}, v \in V : (p, v) \in C \Rightarrow p \in P$.

Thus in an ADO (P, V) all procedures in P use only variables in V and all variables in V are only used by procedures in P. This captures the fact that in an ADO, a set of procedures operates on a set of state variables, while the state variables are invisible outside the ADO. The above definition can be expressed slightly more elegant by introducing some functions.

Definition. (common/used variables/procedures)

1. For $P \subseteq \mathcal{P}$, $cv(P) = \{v \in V \mid \forall p \in P : (p, v) \in C\}$.
   For $V \subseteq \mathcal{V}$, $cp(V) = \{p \in P \mid \forall v \in V : (p, v) \in C\}$.
2. For $P \subseteq \mathcal{P}$, let $uv(P) = \bigcup_{p \in P} cv(\{p\})$.
   For $V \subseteq \mathcal{V}$, let $up(V) = \bigcup_{v \in V} cp(\{v\})$.

In particular, $cv(\{p\})$ (or $cv(p)$ for short) are the variables used by procedure p, and $cp(\{v\})$ (or $cp(v)$ for short) are the procedures which use variable v. $uv(P)$ are *all* variables used by procedures in O, while $cv(P)$ are the *commonly* used variables – *up* and *cp* are to be interpreted analogeously.[1]

Then (P, V) is a module iff $uv(P) \subseteq V$ and $up(V) \subseteq P$ holds. This fundamental closure property for modules reflects both information hiding and low coupling.

Some programming languages permit procedures to be nested. Each local procedure introduces its own set of state variables, which cannot be used by the top-level procedures. Thus, procedure q is local to procedure p iff $cv(p) \subseteq cv(q)$. Sometimes there are not only nested procedures, but also nested ADOs, where the state variables in the inner ADO are not used outside (indeed, C++ supports this kind of nesting).

Definition. Let (P, V) be a module; let $S \subseteq P$. (S, V) is called a submodule of (P, V) iff $uv(P \setminus S) \neq V$.

Hence a local module or submodule consists of a subset of the module's procedures which have additional local variables. Note that $uv(P \setminus S) \subseteq V$ always holds, but that (S, V) is not an ordinary module: $up(V) \subseteq S$ does not hold, as the global variables in V are not only used by S, but also by $P \setminus S$. If S contains only one procedure, this procedure can be considered a local procedure (rather than a local module which contains only one procedure).

In software engineering, cohesion and coupling are important modularization criteria. Cohesion means that the elements of a module are related strongly, while coupling measures interdependence between modules. This motivates the following definitions.

Definition. An ADO (P, V) has *maximal cohesion*, if $\forall p \in P, v \in V : (p, v) \in C$. An ADO has *regular cohesion*, if $\exists \overline{p} \in P \forall v \in V : (\overline{p}, v) \in C$ and $\exists \overline{v} \in V \forall p \in P : (p, \overline{v}) \in C$.

Maximal cohesion means that all procedures use all variables, and all variables are used by all procedures: $cv(P) = V$ and $cp(V) = P$. Regular cohesion means that at least one variable is used by all procedures, and at least one procedure uses all variables: $uv(P) \subseteq cv(\overline{p})$ and $up(V) \subseteq cp(\overline{v})$. Maximal cohesion is almost never found in practice. Even regular cohesion cannot always be identified in existing, well-modularized programs. Both notions are introduced for theoretic reasons.

Definition.

1. Let $P_1, P_2 \subseteq \mathcal{P}$ be two sets of disjoint procedures, let $v \in \mathcal{V}$ be a variable. We say that $P_{1,2}$ are *coupled via* v, iff $v \in uv(P_1) \cap uv(P_2)$.
2. Let $V_1, V_2 \subseteq \mathcal{V}$ be two sets of disjoint variables, let $p \in \mathcal{P}$ be a procedure. We say that $V_{1,2}$ *interfere via* p, iff $p \in up(V_1) \cap up(V_2)$.

This definition means that two sets of procedures (resp. their modules) are coupled if they use the same global variable(s). Similarly, two sets of variables (resp. their modules) interfere, if they are used by the same procedure. Although coupling via global variables is undesirable, in a reengineering setting coupling might be acceptable if there are nested local modules or procedures. Interferences however prevent a modularization, as there is a procedure which uses variables from two different modules – a violation of the information hiding principle.

[1] cv and cp were called σ and τ in [8, 14]; in [18] they are both written as $'$ (derivation). Note that both uv, up are monotone, but both cv, cp are antimonotone.

```
SUBROUTINE R1(...)
COMMON /C1/ V1,V2
...
END

SUBROUTINE R2(...)
COMMON /C2/ V3,V4
COMMON /C3/ V5
...
END
```

```
SUBROUTINE R3(...)
COMMON /C2/ V3,V4
COMMON /C4/ V6,V7,V8
...
END

SUBROUTINE R4(...)
COMMON /C2/ V3,V4
COMMON /C3/ V5
COMMON /C4/ V6,V7,V8
...
END
```

	$V1$	$V2$	$V3$	$V4$	$V5$	$V6$	$V7$	$V8$
$R1$	×	×						
$R2$			×	×	×			
$R3$			×	×		×	×	×
$R4$			×	×	×	×	×	×

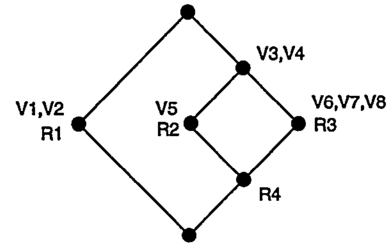


Figure 1: A small source text, its variable usage table and its concept lattice

BASIC NOTIONS OF CONCEPT ANALYSIS

Mathematical concept analysis starts with a relation C between a set of objects $\mathcal{P}$ and a set of attributes $\mathcal{V}$; the triple $\mathcal{C} = (\mathcal{P}, \mathcal{V}, C)$ is called a formal context. In our case, the objects are procedures, and the attributes are global variables.

For any set of procedures $P \subseteq \mathcal{P}$ we can determine their common variables by $cv(P)$. Similarly, for a set of variables $V \subseteq \mathcal{V}$, the common procedures are $cp(V)$. A pair (P, V) where $V = cv(P)$ and $P = cp(V)$ is called a formal concept. Such formal concepts correspond to maximal rectangles in the context table, where of course permutations of rows or columns do not matter. For a concept $c = (P, V)$, $P = ext(c)$ is called the extent and $V = int(c)$ is called the intent of c.[2]

G. Birkhoff discovered in 1940 that the set of all formal concepts for a given formal context c is in fact a complete lattice, the concept lattice $\mathcal{L}(\mathcal{C})$ [1]. The partial order in this lattice is given by $c_1 \le c_2 \iff ext(c_1) \subseteq ext(c_2)$ ($\iff int(c_1) \supseteq int(c_2)$). The infimum of two concepts is computed by intersecting their extents and joining their intents: $c_1 \wedge c_2 = \big(ext(c_1) \cap ext(c_2), cv(cp(int(c_1) \cup int(c_2)))\big)$.[3] The supremum is computed by intersecting the intents and joining the extents of two concepts: $c_1 \vee c_2 = \big(cp(cv(ext(c_1) \cup ext(c_2))), int(c_1) \cap int(c_2)\big)$. Hence the infimum describes the common procedures for two sets of variables, while the supremum describes the common variables for two sets of procedures.

Figure 1 gives a very small example of a formal context and its concept lattice. The context table is generated from a (fictious) FORTRAN source file and captures the use of global variables by subroutines. The labelling of elements allows for an easy interpretation of the lattice; it is achieved as follows. For $p \in \mathcal{P}$, the smallest concept c where $p \in ext(c)$ is $c = sc(p) = \bigwedge\{c \mid p \in ext(c)\}$, and for $v \in V$, the largest concept c where $v \in int(c)$ is $c = lc(v) = \bigvee\{c \mid v \in int(c)\}$. $sc(p)$ is labelled with p, and $lc(v)$ is labelled with v. All concepts greater than $sc(p)$ have p in its extent, and all concepts smaller than $lc(v)$ have v in its intent.[4]

In figure 1, all subroutines below lc(V3) (namely R2, R3, R4) use V3 (and no other subroutines use V3). All variables above sc(R4) (namely V3, V4, V5, V6, V7, V8) are used by R4 (and no other variables are used by R4). Thus the concept labelled R4 is in fact $c_1 = sc(\text{R4}) = (\{\text{R4}\}, \{\text{V3,V4,V5,V6,V7,V8}\})$. The concept labelled V5/R2 is in fact $c_2 = lc(\text{V5}) = sc(\text{R2}) = (\{\text{R2,R4}\}, \{\text{V3,V4,V5}\})$. Hence $c_1 \le c_2$, as c_1 has fewer procedures and more variables. This can be read as an implication: "Any variable used by subroutine R2 is also used by R4". Similarly, $lc(\text{V5}) \le lc(\text{V3}) = lc(\text{V4})$, which translates to "All subroutines which use V5 will also use V3 and V4". The infimum of V5/R2 and V6,V7,V8/R3 is labelled R4, which means that R4 (and all subroutines below sc(R4),[5] but no other) uses both V5 and V6,V7,V8. The supremum of the same concepts is labelled V3,V4, which means that V3 and V4 (and all variables above lc(V2), but no other) are used by both R2 and R3. Such knowledge is not easy to obtain manually from big source files!

The original relation can always be reconstructed via $(p, v) \in C \iff sc(p) \le lc(v)$. Thus formal concept analysis is similar in spirit to Fourier Transformation.

Computation of the lattice has typical time complexity $O(n^3)$ ($n = \max(|P|, |V|)$), but can be exponential in the worst case [18]. In practice, computation of lattices with several hundred elements needs a few seconds on a SparcStation 2.

[2] In fact, cv and cp form a Galois connection, and both $cv \circ cp$ and $cp \circ cv$ are closure operators.

[3] cv and cp are needed because $ext(c_1) \cap ext(c_2)$ can have more attributes than just $int(c_1) \cup int(c_2)$.

[4] In [14, 8, 18] sc and lc are written γ resp. μ.

[5] there are none in the example

	V1	V2	V3	V4	V5	V6	V7	V8	V9	...
R1	×	×	×							
R2	×	×	×							
R3	×	×	×							
R4				×	×	×	×			
R5				×	×	×	×			
R6								×	×	
R7								×	×	
R8								×	×	
R9								×	×	
⋮										

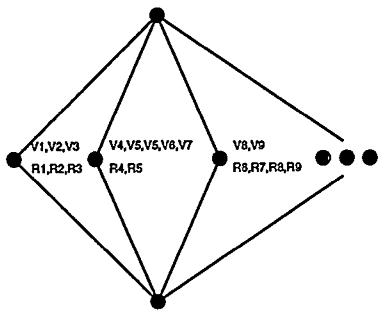


Figure 2: Maximal cohesion corresponds to flat lattices

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12
R1	×	×	×									
R2	×	×	×									
R3	×	×	×	×	×							
R4	×	×	×			×						
R5	×	×	×			×						
R6	×	×	×			×	×	×				
R7	×	×	×			×	×	×				
R8	×	×	×			×			×	×		
R9	×	×	×			×			×	×		
R10	×	×	×			×					×	×
R11	×	×	×			×					×	×

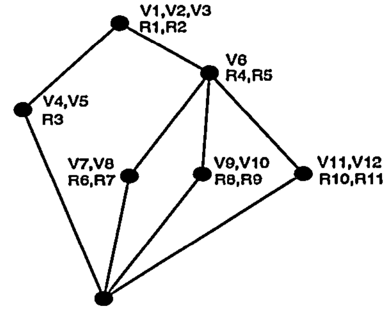


Figure 3: Nested procedures or modules correspond to tree-like lattices

THE CONNECTION BETWEEN MODULES AND CONCEPT LATTICES

Our work is based on the key observation that a module or abstract data object corresponds to a formal concept or a small set of concepts. In this section, we will explain how typical module structures show up in a concept lattice. Later, our insight will be used for reengineering modules from unstructured source code.

Modules with maximal cohesion

We first assume that a program is a collection of modules or ADOs with maximal cohesion. Furthermore we assume there are no nested modules, no global variables, no global procedures. These severe restrictions will be dropped later.

Under the assumption of maximal cohesion, an ADO (P, V) corresponds to a (maximal) rectangle in the variable usage table: $cv(P) = V$ and $cp(V) = P$. Thus a module corresponds to a formal concept of formal context $\mathcal{C} = (\mathcal{P}, \mathcal{V}, C)$. Furthermore, absence of coupling or interferences leads to a particular simple concept lattice $\mathcal{L}(\mathcal{C})$. As there are no procedures which use variables from different ADOs, the intersection of the extents of two ADO's concepts must be empty. Hence the infimum of two concepts must be the bottom element. As there are no variables which are used in different ADOs, the intersection of the intents of two ADO's concepts must be empty. Hence the supremum of two concepts must be the top element. Such lattices are called *flat*. Figure 2 shows a variable usage table and its flat lattice.[6]

[6] Remember that row and column permutations do not influence the lattice.

Nested procedures and modules

For nested procedures or modules, we assume every procedure uses all variables visible to it.[7] Thus, if procedure q is local to procedure p, q's row in the variable usage table contains at least the entries in p's row: $cv(p) \subseteq cv(q) \iff int(sc(q)) \supseteq int(sc(p))$. In the lattice, the corresponding concepts thus form a two-element chain: the "is-local-to"-relationship in the program corresponds exactly to the "is-subconcept-of" relationship in the lattice, as $sc(q) \leq sc(p)$. In particular, variables in the outermost scope show up as labels of the top element. Hence nested procedures produce tree-like concept lattices, which corresponds to traditional nesting hierarchies.[8]

For nested submodules, we also obtain tree-like lattices, because – under the assumption that all procedures use all variables visible to them – the configuration table will contain the same entries for different submodule procedures. If a lattice element is labelled with only one procedure, it corresponds to a local procedure; otherwise, it corresponds to a submodule.

As an example, consider figure 3. Here, we find an ADO M1, with submodules M2, M3, M4, M5, M6, which correspond to the lattice elements $\neq \bot$. M1 consists of procedures R1, R2 and variables V1, V2, V3. M2 adds procedure R3 and variables V4, V5. M3 adds procedures R4, R5 and variable V6 to M1. M4, M5, M6 each introduce two local procedures and variables likewise. Thus M2 and M3 are local to M1, and M4, M5, M6 are local to M3. Note that M2 is a one-row submodule, hence its one and only procedure R3 can as well be considered a procedure local to R1 or R2.

[7] Again, this restriction will be dropped later.

[8] Tree-like lattices are trees with an additional bottom element.

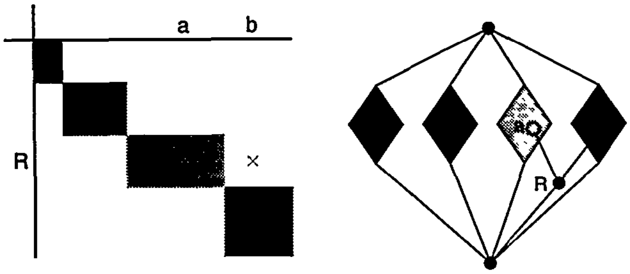


Figure 4: A horizontal decomposition and an interference

Note that the analysis of legacy code may propose a procedure or module nesting which is in contrast to the actual program (for example, FORTRAN does not offer local procedures). It might even be that according to the lattice, a procedure p_2 is considered local and invisible to a procedure p_1, but in the code, p_1 in fact calls p_2. In this case, the lattice shows that the procedure nesting or call graph should be revised, or that there is an implicit hierarchical structure which cannot be expressed syntactically.

Modules with non-uniform variable use

Until now, we have assumed maximal cohesion, which leads to particular simple lattices. In practice, this assumption is of course not true: the lattices obtained from legacy code are much more complicated. In this section, we investigate the effects of non-uniform variable usage in flat module structures. Figure 4 shows a variable usage table which is still segmented into rectangles, but where the rectangles are not completely filled. Instead some entries are missing: not all procedures in an ADO use all ADO's state variables. Such tables produce lattices which are *horizontally decomposable.* The example also contains a simple interference: procedure R uses variables a and b, which are from two different ADOs.

A horizontal decomposition is the inverse to a horizontal sum. The horizontal sum of summand lattices $L_1, L_2, \ldots, L_n$ is $\sum_{i=1}^{n} L_i = \{\top, \bot\} \cup \bigcup_{i=1}^{n} L_i \setminus \{\top_i, \bot_i\}$. That is, the local top and bottom elements are removed from each L_i, and new global top and bottom elements are added. Conversely, a lattice L is horizontally decomposable, if it is a horizontal sum. The module corresponding to a horizontal summand L_i is $(P_i, V_i) = (ext(\top_i), int(\bot_i))$.

Of course, flat and tree-like lattices are horizontally decomposable. Note that for programming languages which enforce encapsulation syntactically, the resulting lattice will always be horizontally decomposable.[9]

[9]As row and column permutations in the table do not matter, horizontal decomposition in the table has exponential complexity, while in the lattice it has only linear complexity.

In legacy code however, modules are not enforced and hence not clearly separated. In particular, there might be interference between module candidates. Interferences can easily be detected in the lattice: procedures that use variables from different module candidates show up as infima between horizontal summands. If the lattice is horizontally decomposable after some interferences have been removed, the system structure is still good.

Horizontal decomposition is achieved by removing top and bottom elements from the lattice graph and determining the connected components; interference detection is based on higher-order graph connectivity. According to the number and "badness" of interferences, the overall quality of the system structure can be measured. [14] and [3] contain a more detailed discussion of horizontal sums and interferences between horizontal summands, and provides numerical measures for the "badness" of an interference.

CASE STUDY 1

Our first small case study is a Modula-2 program from a student project. It serves to illustrate the basic theory, in particular horizontal decompositions. The program is about 1500 lines long and divided into 8 modules; there are 33 procedures which use 16 module variables. The variable usage table was extracted (figure 5), and the corresponding lattice computed (figure 6). The lattice is of course horizontally decomposable[10]. We observe several modules with maximal cohesion (lattice elements 3,4,5,6,7,14,15), a local module containing two procedures (element 2), and a module with neither maximal nor regular cohesion (elements 8,9,10,11,12,13). Note that there are more horizontal summands than modules in the program! Thus the modularization proposal generated from the variable usage does not agree with the actual module structure in the program. Manual inspection confirms that some modules have low cohesion and should be split, and the lattice says which ones.

MODULARIZATION BY INTERFERENCE RESOLUTION

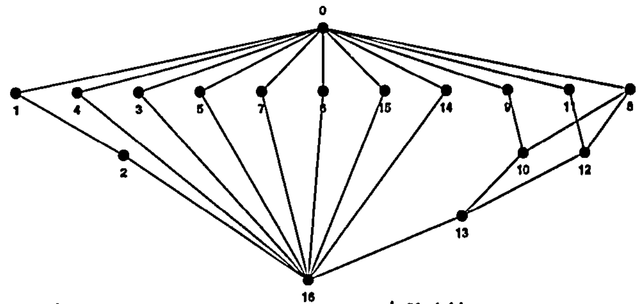
We have seen that horizontal summands are natural module candidates – if the lattice is horizontally decomposable. The i-th horizontal summand generates module $(P_i, V_i) = (ext(\top_i), int(\bot_i))$. In practice, however, legacy code contains interferences. If there are not too many interferences, they can be automatically removed; the source code is transformed accordingly.

The trick for interference resolution is very simple. In functional programming, it is called lambda-lifting. The basic idea is to turn global variables into additional pa-

[10]Remember that for modular languages, this is a consequence of the theory. Unfortunately, Modula-2 allows to export module variables, which can lead to coupling and interferences.

	angelegt	gelöscht	speicherverbrauch	colors	maxstrlength	namehashtab	phonehashtab	hashtabsize	error1	error2	esc
allocate	×										
init	×	×									
analyse	×	×									
initsp			×								
ausgabesp			×								
fuegespein			×								
changeadr				×							
readdata					×						
readline					×						
lookup						×					
exists						×					
rlookup							×				
remove						×	×				
insert						×	×				
calchasvalue								×			
savehashtab						×		×			
partsearch						×		×			
clearhashtabs						×		×			
inithashtabs						×	×	×			
printmessage									×	×	
setbackground											×
settextcolor											×
setattribute											×
clrsrc											×
gotoxy											×

Figure 5: Variable usage table of student Modula-2 program (excerpt)



	Procedures	Variables
0		
1	allocate	angelegt
2	init, analyse	geloescht
3	initsp, ausgabesp, fuegespein	speicherverbrauch
4	ermittelsp, loeschenicht, loeschealles, loesche, einfuegen, erzeuge, elemanz	pointerliste
5	addressinput	done, helpadr, help, eingabe
6	changeadr	colors
7	readdata, readline	maxstrlength
8	lookup, exists	namehashtab
9	rlookup	phonehashtab
10	remove, insert	
11	calchashvalue	hashtabsize
12	savehashtab, partsearch, clearhashtabs	
13	inithashtabs	
14	printmessage	error2, error1
15	setbackground, settextcolor, setattribute, clrscr, gotoxy	esc
16		

Figure 6: Module structure of a student Modula-2 program

rameters. By doing so, they disappear from the variable usage table and become part of the module interface. In the example from figure 1, we can make V5 an additional parameter of procedure R4. Doing so removes the dependency of R4 on V5 from the variable usage table and breaks the edge R2-R4 in the lattice. Afterwards, the lattice is tree-like.

But why not make V6,V7,V8 additional parameters of R4 in figure 1, instead of V5? The reason is that the edge R2-R4 has "weaker coupling power" than R3-R4. This notion can be made precise as follows. Let $c = a \wedge b$ be an interference. If $|int(c)| - |int(a)| > |int(c)| - |int(b)|$, c inherits more variables from b than from a. In this case the connection to a should be broken, as lambda-lifting will add fewer parameters than in the symmetric case. If $|int(c)| - |int(a)| < |int(c)| - |int(b)|$, the edge to b should be broken. This leads to high cohesion. The "weakest coupling" rule can be generalized to interferences of more than two elements.

Formally, an edge $a - c$ is broken as follows. Let $b_1, \ldots, b_k$ be the elements directly above c (thus $a = b_j$). In the configuration table, the set of entries to be removed is then given by $\{(p, v) \mid p \in ext(c),\ v \in int(a) \setminus \bigcup_{i=1, b_i \neq a}^{k} int(b_i)\}$. For each removed entry (p, v), v is made an additional parameter of p.

In figure 1, R4 inherits more variables from R3 than from R2: $|int(sc(\mathrm{R2}))| = 3$, $|int(sc(\mathrm{R3}))| = 5$, while $|int(sc(\mathrm{R2}) \wedge sc(\mathrm{R3}))| = 6$. Therefore, V5 is made an additional parameter to R4 (and not V6,V7,V8). According to the above formula, only the entry (R2,V5) is removed from the configuration table, as V3 and V4 are also in the intent of sc(R3).

Note how the lattice guides restructuring: First, horizontal summands are detected. If the obtained modules are too big, one can apply horizontal decomposition recursively to the summands. If the lattice is not decomposable, interferences will be detected automatically. The algorithm from [14] guarantees that a minimal number of interferences must be removed to make the lattice decomposable, thus minimal changes to the code are required. For each interference, a lambda lifting is proposed in order to resolve it; the "minimum coupling rule" based on the size of the involved intents is used to select the global variables to be transformed into parameters. In figure 4, the analysis will immediately detect the interference and propose to make variable b an additional parameter of procedure R.

It should be pointed out, however, that from a semantic viewpoint more complex transformations (as described e.g. in [5]) might be needed; transforming a global variable into a parameter might be too simple in some situations. Nevertheless, modularization by interference resolution – if possible – is a valuable technique.

MODULARIZATION VIA BLOCK RELATIONS

In this final technical section, we want to propose a more general method for automatic modularization, for which there are no successful case studies at the moment, but which might turn out useful in the future.

Usually procedures do not use all visible variables, while procedures or submodules are nested. For legacy code, this leads to a hierarchy of overlapping sublattices, which prevent horizontal decomposition. The number of interferences often makes their automatic resolution infeasible. In this chapter, we will demonstrate that in some cases, modularization proposals can be generated anyway. The method will only work if regularly cohesive modules can be extracted from the source code.

The basic idea of the method is to determine the *shape* of rectangles in the table, as indicated in figure 4. While non-overlapping shapes lead to horizontally decomposable lattices, overlapping shapes are more complicated to detect. But once a rectangle shape is computed, we can fill in the missing entries and compute a lattice from the "enriched" table. The resulting lattice can be considered a *skeleton* of the original one, as it contains one concept for each original sublattice.

The skeleton of a horizontally decomposable lattice is a flat lattice. Each concept in the skeleton (that is, each rectangle shape in the table) is a candidate for an ADO. Of course, only infima in the skeleton are considered interferences between modules – fine-grained interferences inside a rectangle shape come from non-maximal cohesion and are considered harmless. This is consistent with the modularization method from section 4.

We will now formally define what a rectangle shape is. Due to space limitations, we cannot present the full theory (see [18] for details).

Definition. Let a formal context $\mathcal{C} = (\mathcal{P}, \mathcal{V}, C)$ be given. A ***block relation*** is a formal context $\mathcal{C}' = (\mathcal{P}, \mathcal{V}, C')$ where $C \subseteq C'$, and for $p \in \mathcal{P}$, $cv_{\mathcal{C}'}(p)$ is an extent in $\mathcal{L}(\mathcal{C})$, and for $v \in \mathcal{V}$, $cp_{\mathcal{C}'}(v)$ is an intent in $\mathcal{L}(\mathcal{C})$.

The three conditions together make sure that a block relation is indeed the shape of a rectangle in the original table. The sides of such a rectangle are extents resp. intents, thus they must either occur as horizontal or vertical "lines" in the original table, or be suprema/infima in $\mathcal{L}(\mathcal{C})$ of such "elementary" rectangles. This explains why at least elementary rectangle shapes correspond to modules with regular cohesion, while these can be combined to bigger modules without regular cohesion.

Block relations can also be characterized through concept lattices via the following isomorphism theorem:

Theorem. [18] Let $\mathcal{C}'$ be a block relation to $\mathcal{C}$. Then $\mathcal{L}(\mathcal{C}') \simeq \mathcal{L}(\mathcal{C})/\Theta$, where Θ is a reflexive and symmetric relation on the lattice elements which is compatible with supremum and infimum. Each block of $\mathcal{C}'$ corresponds to a Θ-class.

If Θ is also transitive, it is a lattice congruence. It is remarkable that the factor lattice $\mathcal{L}(\mathcal{C})/\Theta$ exists even for non-transitive "congruences". It is even more remarkable that the Θ-classes correspond to rectangle shapes.

	$V1$	$V2$	$V3$	$V4$	$V5$	$V6$
$R1$	•	×	×	×	×	×
$R2$	×	•	×	×	•	×
$R3$	×	•	×	•	×	×
$R4$	×	×	×	•	•	×
$R5$		×	•	×	×	×
$R6$		•		×	×	•

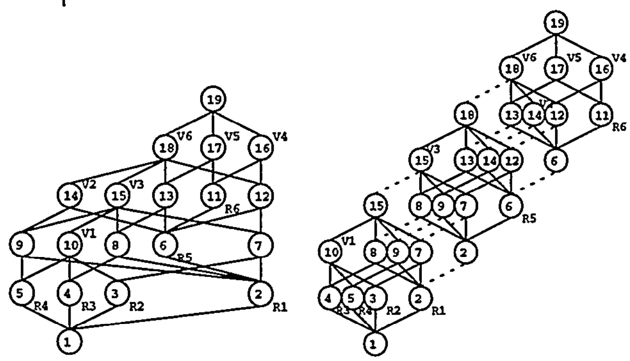


Figure 7: A context table, its lattice, a block relation, and its corresponding Θ-classes

Note that the set of block relations (resp. their "congruences") form itself a lattice, and that there is an algorithm to effectively compute all block relations for a given table. For any block relation resp. its corresponding Θ, its skeleton is just $\mathcal{L}(\mathcal{C})/\Theta$.

As an example, consider figure 7. A table and its lattice are shown; the lattice does not reveal any modularization proposal at a first glance. The table contains also a block relation, which consists of the original ×-entries and the additional •-entries. The bullets have been chosen such that the block relation consists of only three blocks (i.e. three rectangle shapes in the original table) and corresponds to a skeleton which is a three-element-chain. This indicates there are three module candidates. Figure 7 (right part) displays an isomorphic copy of the lattice, but now the skeleton and the Θ-classes which correspond to the rectangle shapes are visible.

Thus the modularization proposal consists of three modules. The first module corresponds to the top tolerance class. It contains procedure R6 and variables V2, V4,V5,V6. Indeed, there is a corresponding rectangle shape in the table. The next module is local to the first one. It corresponds to the middle tolerance class and introduces local procedure R5 and local variable V3. The last submodule corresponds to the bottom tolerance class and introduces procedures R1, R2, R3, R4 as well as local variable V1. In this fictious example, the resulting module structure is interference free.

In case there is more than one block relation, the restructurer must decide which one is best. Note that a generalization of the method to non-regular cohesive modules is not known today, and is unlikely to exist.

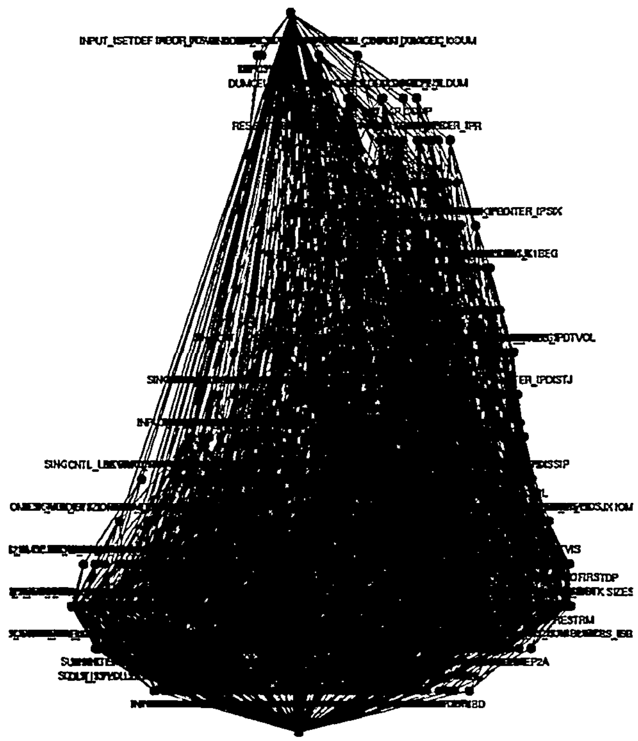


Figure 8: Variable usage structure of aerodynamics system

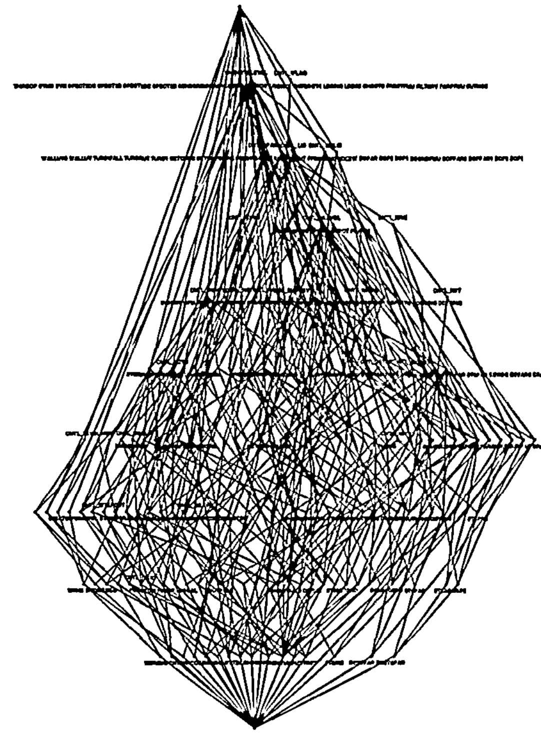


Figure 9: Variable usage structure for COMMON block "CNTL"

CASE STUDY 2

Our next real-world example is a legacy code written in FORTRAN. The program is an aerodynamics system used for airplane development in a national research institution. The system is about 20 years old, and has undergone countless modifications and extensions. The source code is 106000 lines long, consists of 317 subroutines, and uses 492 global variables in 46 COMMON blocks. One of the goals of the analysis is to reshape COMMON blocks such that each ADO corresponds to one COMMON block. Several manual restructuring efforts had not been very successful, so it was decided to try concept analysis.

After the variable usage table was built, the lattice was constructed[11]. It contains no less than 2249 elements! The number of elements in itself is not the problem (after all, it is a large program), but unfortunately the lattice is so full of interferences that it is impossible to reveal any structure (figure 8). There is no way to make the lattice horizontally decomposable by removing just a small number of interferences.

[11]this required 11 seconds on a SparcStation20

Several experiments tried to analyse just part of the system. The program contains a particularly intricate COMMON block called "CNTL", which contains 26 variables. These variables are used in 192 subroutines, and the resulting lattice does not look very encouraging either: it has 194 elements (figure 9). Another experiment examined the "OUTPUT"-subsystem, which consists of 50 subroutines using 278 global variables from 26 COMMON blocks; the resulting lattice still has 259 elements and is full of fine-grained interferences.

We also tried to determine block relations. Unfortunately, neither the lattice for the whole system nor the lattice for the "CNTL" COMMON block had usable block relations, hence no automatic modularization was possible. We also tried to apply subdirect decomposition [16] and subtensorial decomposition [17] to the lattice, as described in [3]. These decomposition techniques are motivated by algebraic rather than software engineering issues, and failed also. Thus our technique was no more successful than previous efforts on the system.

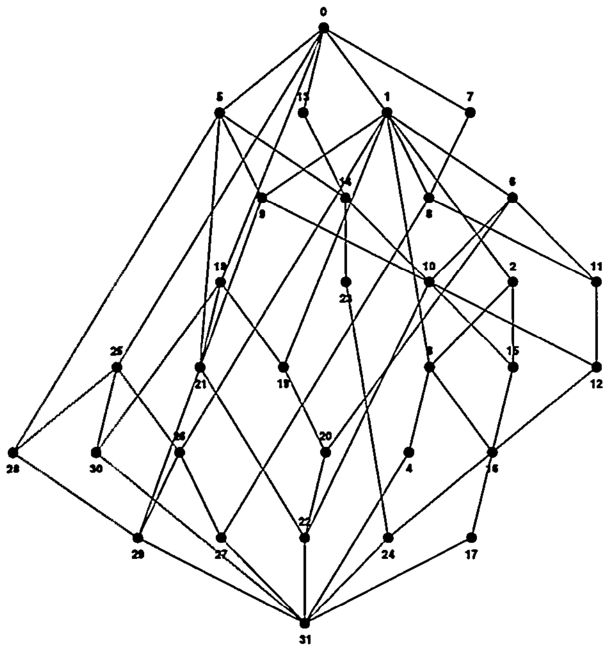


Figure 10: Variable usage structure of a commercial COBOL program

CASE STUDY 3

Our final example is a system written in COBOL, namely an accounting system developed for a North-German car manufacturer. We have analyzed two programs of this relatively new system.

The first program contains about 500 executable statements. In COBOL, there are no procedures, but there are so-called sections, which are a kind of parameterless procedures. Hence the relation between sections and variables was analysed, there were 11 sections and 88 variables (the variables being complex records). Figures 10 and 11 show the result: a lattice with 32 elements. The lattice is not horizontally decomposable, and there are too many interferences to try automatic interference resolution. In fact, the interference detection algorithm found several simple interferences, but their removal did not produce horizontal summands. For example, removal of element 8 (interference between elements 1 and 7) isolates element 7, but this does not lead to a usable module. The lattice has no block relations either. Still, one could argue that this program is too small to be modularized, and that all the "interferences" just demonstrate high cohesion.

We therefore tried a larger program of the same system, consisting of more than 5000 source lines. It contained 165 variables and 44 sections. The resulting lattice has 144 elements (figure 12). Again, countless interferences and missing block relations prevent automatic modularization. Another, even larger program produced a lattice with several hundred elements and was not decomposable either. As the system is only a couple of years old, we suspect it to be characteristic for contemporary COBOL programming style. Note the numbered variable names in figure 11, and note the number of global variables in both programs!

	Variables	Sections
0		
1	W8-IOPCB, W3-ZBER, S98-W00-PROTOKOL, S96-P00-BATINFO, S22-P30-KONTFGRU, S22-P22-BANKBDC, S22-P21-BANKIDC, S22-P20-ZAHLPRUE, S22-P10-LIEFUSER, S21-P30-LIEFFIRM, S21-P20-LIEFNAME, S07-P10-BDLIEFFI, S06AX-P10K-KEY, S06-P10-BILIEFFI, S02AX-P20K-LIEFER, S02AX-P20K-KEY, S02AX-P20K-FIRMNR, S02-P20-KFGLIFI, A96-P00U-BATINFO, A22-P30U-KONTFGRU, A22-P22U-BANKBDC, A22-P21U-BANKIDC, A22-P20U-ZAHLPRUE, A22-P10U-LIEFUSER, A22-P00U-LIEFFIRM, A21-P20Q-LIEFNAME, A21-P30U-LIEFFIRM, A07-P00Q-BANKBDC, A07-P10U-BDLIEFFI, A06AX-P00Q-KEY, A06-P10U-BILIEFFI, A06-P00Q-BANKIDC, A02-P20U-KFGLIFI,	A512-AUFNEHMEN
2		A21-EINGABE
3		
4	Z1-ZAEHLER	U01-SATZ-ZAEHLER
5	S05BAX-P00D-KTOTYP, S05AAX-P00K-KEY, S05-P00-FIRMREF, S03-P00-FIRMA, S02-P00-KONTFGRU, EX1FIRMA, EX1-FIRMA, A05AX-P00Q1-SATZART, A05AX-P00Q1-RESTKEY, A05AX-P00Q1-KEY, A05AX-P00Q1-FIRMNR, A05-P00Q1-FIRMREF, A03AX-P00Q-FIRMNR, A03-P00Q-FIRMA	A10-VORLAUF
6	S22-P00-LIEFFIRM, A22-P00Q-LIEFFIRM	A511-AUFNEHMEN
7	EW1AX-LIEFER	A51-VERARBEITEN
8	S21-P00-LIEFER	
9	P2-XK901A, A02AX-P00Q-KONTFGRU, A02AX-P00Q-FIRMNR, A02-P00Q-KONTFGRU	
10		
11	A21-P00Q-LIEFER	
12	W2-FIRMA, K1-CALL-FKT	
13	RETURN-CODE	A90-ABSCHLUSS
14		
15	W0-FILE-STATUS, EW1LIEF	
16		
17	U1-SCHALTER	A01-GESAMT
18		U90-DBSWITCH
19	LXK96-XKA96P	
20	LXK22-XKA22P	
21	LXK05-XKA05S, LXK05-XKA05P	
22	P49-XK949A	
23	PROBA-ID	A91-ABBRUCH
24	P1-ABEND	
25		A00-BASIS
26	LXK98-XKA98P, LXK07-XKA07P, LXK06-XKA06P	
27	LXK21-XKA21P	
28	LXK03-XKA03P	
29	LXK02-XKA02P	
30	LXKNN-XKC96P, LXKNN-XKC22P, LXKNN-XKC05S, LXKNN-XKC05P, LXKNN-XKB96P, LXKNN-XKB22P, LXKNN-XKB05S, LXKNN-XKB05P, LXKNN-XKA96P, LXKNN-XKA22P, LXKNN-XKA05S, LXKNN-XKA05P	
31		

Figure 11: Labels for lattice in figure 10

RELATED WORK

Methodologies and tools for reverse engineering and restructuring differ with respect to the source information they use and the program transformations they apply. Cimitile and Visaggio [2] analyse the call graph of a system. The directed call graph induces a call dominance tree which is interpreted as a functional dependency graph which in turn is used to generate module candidates. Different edges in the tree are interpreted as *part of* and *uses* relationship, and modules containing

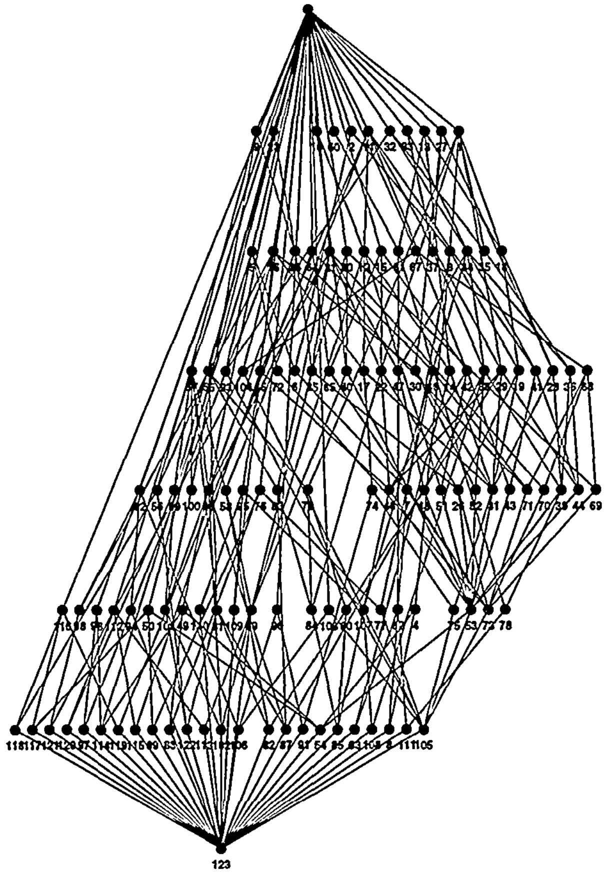


Figure 12: Another program of the same COBOL system

functions are derived accordingly. This approach differs from ours in that it uses the *calls function*-relation while we rely entirely on the *uses variable*-relation.

The work of Schwanke [13] uses a broader source of information to cluster procedures into modules: all functions get so-called features attached, which are names of non-local procedures, types, variables, or type definitions. He defines a similarity measure for procedures derived from these features and clusters procedures according to their similarity. Patel et al. [11] generates a multi-dimensional vector based on a procedure's non-local accesses to objects, and measures similarity by the angle (scalar product) between these vectors.

Note that Schwanke's similarity measure contains free parameters and therefore requires tuning, while Patel's approach does not use any heuristics. Our approach is even more deterministic: the raw data can always be reconstructed from the analysis results.

Müller et al. [9] consider resources being *required* by procedures and *provided* by others. Similar to Schwanke they consider procedures, constants, and variables as resources that form a resource relation. Additionaly they take into account how the system is organized into files and directories. Both relations induce directed, layered graphs. In order to assess coupling and cohesion between nodes of the resource relation, the exchange of resources between clients and providers is measured, as well as the set of common suppliers and clients. The editor part of the Rigi system can be used to optimize coupling and cohesion manually, but it does not automatically generate module candidates.

The tools built around star diagrams [6, 7] use additional data flow analysis to track down the usage of (complex) variables. The idea is to collect all parts of a program that (indirectly) use a certain variable, and apply restructuring transformations in order to build a new abstract data type. This leads to graphs, called star diagrams which are similar to program slices. The user has to select the variable that should be investigated, and to choose a restructuring transformation [5] to be applied to this variable. Star diagrams, other approaches based on program slicing [4, 10], and our approach have in common that they entirely rely on the usage of variables. This is in contrast to the above approaches which also consider procedure calls.

To discover modular structure, our method should be applied together with other methods. Empirical studies must show how concept analysis compares with these methods. We believe that complex reengineering tasks cannot be tackled with one method alone, but that in practice a method mix will be required – in particular if semiautomatic modularization is to be achieved.

FUTURE WORK

Basic mathematical concept analysis, as used in this article, is not "continuous": a single "wrong" entry in the variable usage table can destroy decomposition properties of the lattice. This behaviour seems to prevent automatic modularization in many cases. A more realistic restructuring approach must probably include some heuristics. For example, we can measure *how often* a procedure p uses a variable v. The resulting table contains integer values instead of booleans, and allows to discover "mavericks" ([13]) by comparing usage numbers to a treshold value. This simple technique will reduce the lattice, which is bad for analysis purposes (loss of information), but good for lattice decomposition.

But there are mathematically deeper ways to handle numerical tables. Recently, fuzzy concept analysis has been developed [12]. This approach merges fuzzy set theory and concept analysis. Table entries are no longer boolean values, but numbers between 0 and 1; $T[o, a] = x$ means that object o has attribute a to a degree of $x \cdot 100$ percent. We will investigate whether fuzzy concept analysis has more continuous behaviour and allows for easier lattice decomposition and scaleable restructuring.

CONCLUSIONS

"Die Grenzen meiner Sprache sind die Grenzen meiner Welt"[12] said Ludwig Wittgenstein, and our case studies show that he was perhaps right. While Modula-2 programs lead to decomposable lattices, a variety of FORTRAN and COBOL programs revealed no modular structure at all. Hence our modularization method could not be applied to the two legacy systems we have examined. Automatic modularization is possible only if there is still some hidden structure, but fails on software which is near entropy death.

Indeed, we do not consider our method to be fully automatic: it should be considered and used as an intelligent assistant. In many cases, more substantial changes to source code will be necessary than just reshaping COMMON blocks or turning global variables into interface parameters.

Still, mathematical concept analysis is a valuable tool to assess modular structure. It not only determines fine-grained dependencies between procedures and variables, but also can be used to assess the overall quality of a software system with respect to coupling, cohesion and interferences. Future work must show whether automatic restructuring based on concept analysis can be achieved.

Acknowledgements

Uwe Tapper implemented the FORTRAN frontend. Bernhard Ganter provided technical support for the block relation algorithm. Andreas Zeller provided helpful comments on a preliminary version of this article. Susan Deikman pointed out that "chemotherapeutics is useful even though it does not cure every cancer".

The work described in this paper is part of the inference-based software environment NORA[13].

REFERENCES

[1] G. Birkhoff: Lattice Theory. American Mathematical Society, Providence, R.I., 1st edition, 1940.

[2] A. Cimitile, G. Visaggio: Software salvaging and the call Dominance Tree. J. Systems Software (28): 117–127, 1995.

[3] P. Funk, A. Lewien, G. Snelting: Algorithms for concept lattice decomposition and their application. Report 95-09, Computer Science Department, Technische Universität Braunschweig, 1995.

[4] K. Gallagher, J. Lyle: Using program slicing in software maintenance. IEEE Transactions on Software Engineering 17,8 (August 1991), pp. 751–761.

[12] "The limitations of my language are the limitations of my world"

[13] NORA is a drama by the Norwegian writer H. IBSEN. Hence, NORA is no real acronym.

[5] W. G. Griswold, D. Notkin: Automated assistance for program restructuring. ACM Transactions on Software Engineering and Methodology 2,3 (July 1993), pp. 228–269.

[6] R. Bowdidge, W. G. Griswold: Automated support for encapsulating abstract data types. Proc. 2nd Int. Conf. on the Foundations of Software Engineering, pp. 97–110, ACM 1994.

[7] W. G. Griswold, M. L. Chen. et al.: Tool support for planning the restructuring of data abstraction in large systems. Int. Conf. on the Foundations of Software Engineering, pp. 33–45, ACM 1996.

[8] M. Krone, G. Snelting: On the inference of configuration structures from source code. Proc. 16th International Conference on Software Engineering, Mai 1994, IEEE Comp. Soc. Press, pp. 49-57.

[9] H.-A. Müller, M. A. Orgun, et al.: A reverse engineering approach to subsystem structure identification. Software Maintenance: Research and Practice, 5(4): 181–204, 1993.

[10] L. Ott, J. Thuss: Slice based metrics for estimating cohesion. Proc. IEEE-CS International Metrics Symposium (1993), pp. 78 – 81.

[11] S. Patel, W. Chu, R. Baxter: A measure for composite module cohesion. Proc. 14th International Conference on Software Engineering, 1992, IEEE Comp. Soc. Press, pp. 38-48.

[12] S. Polland: Fuzzy Begriffe – formale Begriffsanalyse unscharfer Daten. Springer Verlag 1997.

[13] R. Schwanke: An intelligent tool for reengineering software modularity. Proc. 13th International Conference on Software Engineering, 1991, IEEE Comp. Soc. Press, pp. 83-92.

[14] G. Snelting: Reengineering of configurations based on mathematical concept analysis. ACM Transactions on Software Engineering and Methodology 5,2 (April 1996), pp. 146-189.

[15] R. Wille: Restructuring lattice theory: an approach based on hierarchies of concepts. In: I. Rival, (Ed.), Ordered Sets, pp. 445-470, Reidel 1982.

[16] R. Wille: Subdirect decomposition of concept lattices. Algebra Universalis 17 (1993), pp. 275-287.

[17] R. Wille: Tensorial decomposition of concept lattices. Order 2 (1985), pp. 81-95.

[18] B. Ganter, R. Wille: Formale Begriffsanalyse – Mathematische Grundlagen. Springer Verlag 1996.